Échanges non bloquants de données ordonnées entre producteurs multiples et consommateur unique

Paul Godard

▶ **To cite this version:**

Paul Godard. Échanges non bloquants de données ordonnées entre producteurs multiples et consommateur unique. COMPAS 2019 - Conférence d'informatique en Parallélisme, Architecture et Système, Jun 2019, Anglet, France. hal-02381769

HAL Id: hal-02381769
https://hal.science/hal-02381769

Submitted on 26 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compas'2019 : Parallélisme / Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

Échanges non bloquants de données ordonnées entre producteurs multiples et consommateur unique

Paul Godard *

Société Caldera - 1 Rue des Frères Lumière, 67201 Eckbolsheim, France
Université de Strasbourg - ICube - Équipe ICPS
INRIA Grand Est
paul.godard@caldera.com

Résumé
La parallélisation d'un travail sur plusieurs unités de calcul entraîne la dispersion des résultats produits et nécessite généralement de les réunir afin de continuer l'exécution du programme. L'échange unidirectionnel de données en résultant peut être décrit par la relation producteur - consommateur. Dans ce papier, nous nous intéressons à l'échange de données ordonnées de taille fixe entre un ou plusieurs producteurs vers un unique consommateur, dans un contexte distribué ou non. Nous proposons une solution permettant d'optimiser le débit de ces échanges en favorisant une écriture non bloquante pour les producteurs ainsi qu'une gestion de la mémoire et une synchronisation adaptée à cet objectif. Notre solution est évaluée selon plusieurs configurations et se montre jusqu'à 6,6 fois plus performante qu'une approche avec synchronisations naïves.

Mots-clés : producteur consommateur, synchronisation non bloquante, flux de données

---

1. Introduction

L'informatique distribuée et la démocratisation toujours grandissante de la parallélisation sur les architectures *multi-core*, *many-core* et *accélérateurs* a poussé à la conception de nombreux patrons de conception permettant une interconnexion efficace des différentes unités de calcul impliquées. Formalisée depuis les années 1960, la relation producteur - consommateur reste encore aujourd'hui une architecture incontournable de par la simplicité et l'universalité de l'interaction qu'elle décrit. Cette relation définit une séparation stricte des rôles de part et d'autre d'un échange unidirectionnel de données entre une ou plusieurs unités de calcul productrices de données et une ou plusieurs unités de calcul consommatrices des données produites. La parallélisation des traitements sur plusieurs producteurs est courante lorsqu'il est nécessaire d'atteindre un débit élevé. Ainsi, elle est mise en œuvre dans différents contextes : aussi bien entre les *threads* d'un même processus, qu'à la mise en relation de noeuds d'une grande ferme de calcul, en passant par l'exploitation de cartes accélératrices.
La topologie, le nombre de producteurs et de consommateurs impliqués ainsi que les données échangées peuvent différer et présenter des contraintes spécifiques à chaque domaine. Dans

∗. Remerciements à mes encadrants Vincent Loechner et Cédric Bastoul pour leur relecture, ainsi qu'à Ervin Altıntaş (Stagiaire Master 2 en 2018) pour ses travaux préparatoires.

ce papier nous nous concentrons sur l'architecture courante impliquant un ou plusieurs producteurs qui délivrent des données ordonnées de taille fixe à un unique consommateur. Ces données, ordonnées grâce à une numérotation continue, peuvent être produites en parallèle par les différents producteurs et ainsi soumises de façon désordonnée au consommateur bien que celui-ci doive les traiter dans l'ordre. Réaliser l'envoi de données au consommateur de façon désordonnée permet de ne pas créer de dépendance entre les producteurs, ce qui procure de nombreux avantages : libérer rapidement de l'espace mémoire chez le producteur quand son traitement est terminé, exploiter un bus ou un lien réseau actuellement inutilisé, réaliser les envois en parallèle, etc. Cette architecture est notamment présente dans les systèmes de traitement parallèles de flux multimédia, comme la génération ou le traitement d'images, de vidéos ou de sons, dans lesquels les données issues des différentes unités de calculs doivent ensuite être réordonnées.
La solution présentée dans ce papier a pour objectif d'offrir un haut débit d'échange de données entre les producteurs et le consommateur. Pour ce faire, nous proposons un mécanisme de synchronisation et de gestion des données en mémoire permettant d'éviter la coûteuse duplication des données entre un producteur et le consommateur, ainsi que minimisant l'occurrence des opérations bloquantes et les délais d'attente si celles-ci surviennent. De plus, notre solution permet de synchroniser différentes architectures, qu'elles soient parallèles et/ou distribuées. Pour cette dernière, nous considérons les producteurs comme des intermédiaires locaux gérant la communication (*socket* ou *bus PCIe*) avec l'unité de calcul distante.

2. Problématique et solution apportée

La conception d'un système efficace d'échange de données ordonnées entre des producteurs et un consommateur indépendants pose une problématique qui s'articule autour de deux aspects principaux : le stockage en mémoire des données échangées et la synchronisation des accès à cette mémoire.

2.1. Mémoire

L'architecture actuelle des systèmes implique des mécanismes de protection des données entre différents processus. En effet, hors processus s'exécutant en mode noyau, les espaces d'adressage de chaque processus ne sont accessibles que par leurs *threads* et non par les autres processus du système. Cette sécurité empêche l'écriture directe de données de la part d'un producteur dans l'espace mémoire d'un consommateur et inversement pour une lecture directe depuis ce dernier dans l'espace mémoire d'un producteur si ceux-ci sont des processus indépendants. Il convient donc d'utiliser un mécanisme explicite de partage de données entre processus tout en limitant leur recopie entre le producteur et le consommateur, sous peine d'affecter fortement les performances de la solution. La solution doit également tenir compte de l'environnement dans lequel elle s'exécute afin de respecter ses contraintes. Il s'agit en particulier de ne pas requérir davantage de mémoire que le système peut en fournir, en incluant celle nécessaire à l'exécution simultanée du ou des producteurs et consommateurs. De plus, dans le cadre d'échanges indépendants de données ordonnées, il doit être possible de supporter des données temporairement manquantes afin de ne pas linéariser les échanges. La complexité de cette condition est assouplie dans notre cas d'étude puisque nous nous intéressons à l'échange de données de taille fixe, ce qui permet de faciliter la prédiction de la quantité de mémoire nécessaire à chaque donnée ainsi que son emplacement futur en mémoire.
Afin de satisfaire ces différentes contraintes, notre solution s'appuie sur l'utilisation de *mémoire partagée* comme zone de transfert des données d'un processus producteur au processus

consommateur puisqu'elle permet des accès en écriture et en lecture à des processus indépendants. Bien que le noyau Linux propose trois interfaces différentes à la création de ces mémoires partagées (System V, POSIX et memfd), nos expérimentations préalables ont montré des performances identiques, suggérant l'utilisation d'un même sous-système de partage de mémoire inter processus. Notre solution utilise l'interface POSIX de façon à maximiser sa portabilité. Nous organisons cette mémoire partagée sous la forme d'un *buffer circulaire* permettant de stocker de manière cyclique des données dans une plage d'adresses fixes. La quantité de mémoire partagée utilisée par notre solution est paramétrable par l'utilisateur lors de son initialisation en indiquant la taille d'une donnée et le nombre de données qu'il souhaite pouvoir y stocker simultanément. Si l'utilisateur sélectionne une capacité de stockage supérieure à une seule donnée, alors notre solution permettra l'écriture parallèle et non bloquante d'autant de données, à la condition que leur emplacement (calculé à partir du numéro de la donnée et de leur taille) soit libre, c'est-à-dire ne contienne pas une donnée non lue par le consommateur.

L'utilisation de mémoire partagée nous permet également de bénéficier d'un mécanisme de *zero-copy* entre les producteurs et le consommateur. Pour ce faire, l'interface de programmation (API) de notre solution retourne directement l'adresse mémoire des données, ainsi le programme peut directement utiliser cette adresse dans ses opérations d'écriture et de lecture. Une fois l'opération effectuée il indique, via l'API, que cette donnée est désormais lisible ou bien réutilisable pour une autre écriture. L'obtention de ces adresses, qui déclenche la possibilité d'y écrire ou d'y lire, est assurée par le mécanisme de synchronisation.

2.2. Synchronisation

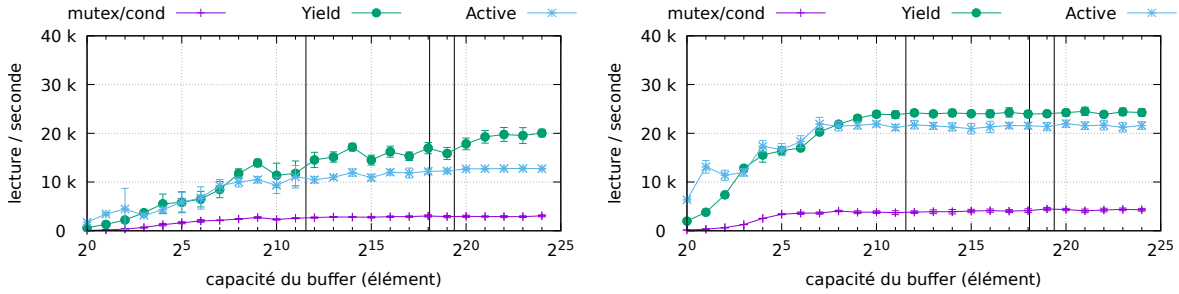
La synchronisation des producteurs et du consommateur est nécessaire pour deux opérations :

— pour qu'un producteur obtienne l'adresse de la donnée qu'il veut écrire ;
— pour que le consommateur obtienne l'adresse de la donnée suivante à lire.

L'échange de données étant dépendant de l'exécution du ou des producteurs, il existe deux scénarios empêchant d'honorer immédiatement une requête : la lecture d'une donnée encore non écrite ; et l'écriture d'une donnée alors que l'emplacement est utilisé par une donnée non lue. De façon à simplifier son utilisation, notre solution propose de bloquer le processus appelant afin de le mettre en attente avec pour objectif de le débloquer avec une latence minimale.

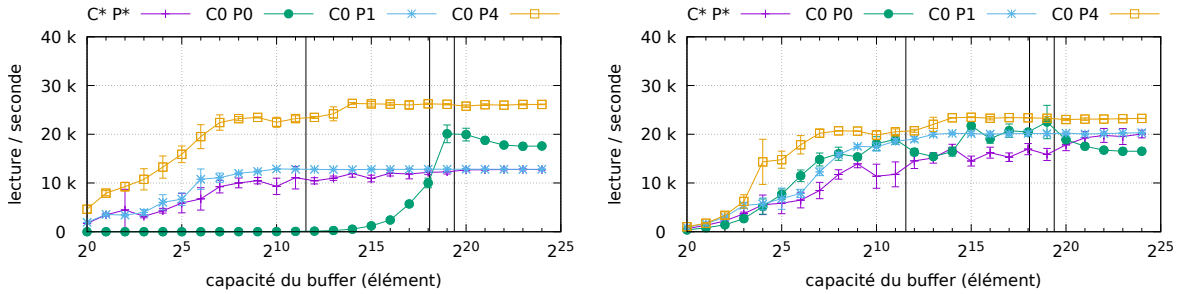
Une approche naïve consiste à assurer l'intégrité de l'état (lu/non lu) des données du buffer circulaire en assurant l'accès à une section critique via des mutex puis si nécessaire de suspendre/réveiller les processus via l'utilisation de conditions et de signaux. Ces opérations sont classiquement réalisées via l'utilisation de la bibliothèque `pthread`. Cette approche naïve que nous évaluons dans la section 3 y est désignée sous le nom de *mutex/cond*.

Pour accélérer cette synchronisation, notre solution propose de supprimer le recours à des sections critiques. Ainsi, pour connaître l'état d'utilisation du buffer circulaire notre solution place en en-tête du buffer circulaire deux informations : un entier désignant le prochain numéro de données que le consommateur doit lire (initialement 1) ; et un tableau d'entiers dont chaque élément indique le numéro (initialement 0) de la donnée actuellement stockée à l'adresse correspondante dans le buffer circulaire. Ainsi, l'état (c'est-à-dire le numéro) de chaque donnée peut être interrogé et modifié sans jamais être corrompu. Si la valeur du prochain élément à lire et l'état de la donnée correspondante tombent dans un des deux scénarios bloquants présentés précédemment, alors notre solution propose deux stratégies pour maintenir efficacement le processus en attente. La première consiste en une attente active sur la valeur de l'état, jusqu'à ce que celui-ci obtienne la valeur attendue. La seconde ajoute à cette attente active un appel système indiquant au noyau le souhait d'être préempté par un autre processus. En section 3 ces deux stratégies sont respectivement évaluées sous les noms d'*active* et de *yield*.



(a) *Intel Xeon D-1521* (b) *Intel Core i5-3470*

Figure 1 – *Nombre de lectures par seconde pour **1** producteur en fonction de la capacité du buffer et de la méthode d'attente utilisée sur différents processeurs.*



(a) *Attente active* (b) *Attente yield*

Figure 2 – *Nombre de lectures par seconde pour **1** producteur en fonction de la capacité du buffer et du placement des processus selon la méthode d'attente utilisée sur Intel Xeon D-1521.*

3. Expériences

3.1. Mesures

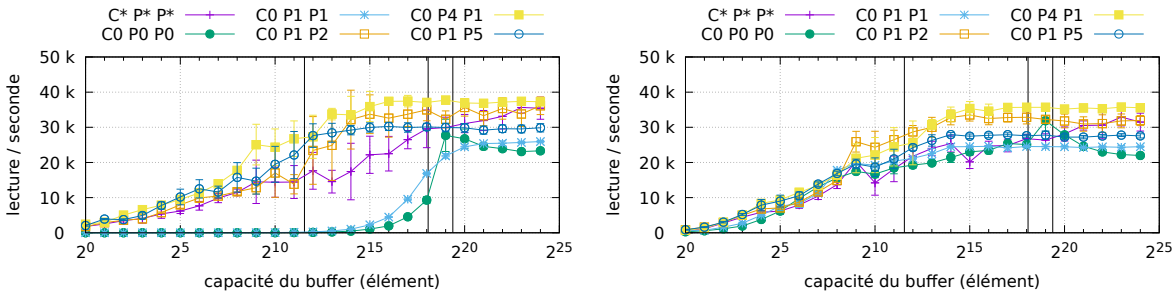
Afin d'évaluer les performances de notre solution, nos expériences mesurent le temps d'exécution du mécanisme de synchronisation en incluant l'accès aux données échangées. Afin d'isoler au mieux le temps pris par les mécanismes, la taille des éléments est définie à 1 octet. Le fonctionnement de la synchronisation étant dépendante du niveau de remplissage du buffer (un faible remplissage augmente le risque d'attente pour la lecture tandis qu'un fort remplissage celui d'attente pour l'écriture) nous évaluons notre solution en fonction de la capacité du buffer. La notion de circularité impliquant des mécanismes de bordure, chaque expérience réalise l'échange d'une quantité d'éléments supérieure à la capacité du buffer.

Notre programme d'évaluation exécute un *thread* consommateur qui lit en continu le buffer circulaire : dès qu'il reçoit l'adresse de l'élément suivant, il y accède puis le signale comme lu ; parallèlement, un ou plusieurs *threads* producteurs demandent en continu d'écrire dans le buffer circulaire en entrelaçant leurs écritures selon le nombre de producteurs.

Nous réalisons nos mesures sur deux processeurs différents : un Intel Xeon D-1521 ayant 4 cœurs physiques **avec** *hyper-threading* cadencés à **2,40 GHz** ; un Intel Core i5-3470 ayant 4 cœurs physiques **sans** *hyper-threading* cadencés à **3,20 GHz**. Nos programmes sont compilés avec `gcc` en version 8.2.0 et exécutés sous Ubuntu 18.04.1 LTS avec le noyau Linux 4.15.0-43-generic.

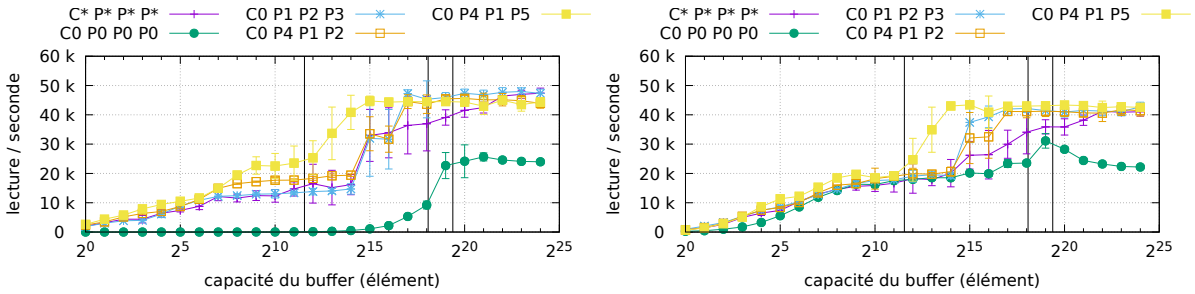
3.2. Résultats obtenus

Les résultats de nos expériences sont présentés dans les figures 1 à 4. Sur chacune, les marqueurs verticaux représentent le nombre d'éléments à partir duquel la taille du buffer circulaire



(a) *Attente active* (b) *Attente yield*

Figure 3 – *Nombre de lectures par seconde pour* **2** *producteurs en fonction de la capacité du buffer et du placement des processus selon la méthode d'attente utilisée sur Intel Xeon D-1521.*



(a) *Attente active* (b) *Attente yield*

Figure 4 – *Nombre de lectures par seconde pour* **3** *producteurs en fonction de la capacité du buffer et du placement des processus selon la méthode d'attente utilisée sur Intel Xeon D-1521.*

dépasse celle du cache, respectivement de gauche à droite celle du cache L1, L2 et L3.
La figure 1 met en avant les faibles performances d'un système de synchronisation *mutex/cond* pthread par rapport à notre solution, en effet, avec la stratégie de synchronisation *yield* notre solution exécute jusqu'à 6,6 fois plus de lectures par seconde. Il est intéressant de remarquer la meilleure performance de la solution *active* par rapport à la solution *yield* pour les buffers de petites capacités (< 8 éléments). En effet, la faible capacité du buffer provoquant de nombreux blocages, celle-ci se montre plus réactive.
Les figures 2, 3 et 4 comparent les performances des méthodes de synchronisation en fonction du nombre de producteurs et de l'assignation explicite ou non des producteurs et du consommateur sur les cœurs du CPU. La numérotation des cœurs correspond à celle du système (via `/proc/cpuinfo`), le cœur 4 est ainsi le cœur *hyperthreadé* complémentaire du cœur 0. La notation dans les légendes des figures indique à quel numéro de cœur le consommateur (C) et les producteurs (P) sont assignés. Une assignation explicite des producteurs et du consommateur sur les cœurs du CPU permet d'influer fortement sur les performances par rapport à l'assignation dynamique faite par le noyau par défaut (annotée $*$). Nos expériences n'ayant pas montré d'améliorations notables des performances pour la synchronisation *mutex/cond* en fonction de l'assignation sur les cœurs, nous ne la présentons pas dans ces figures.
Le seul choix d'assignation clairement préjudiciable consiste à assigner tous les processus sur le même cœur. Cela est particulièrement visible pour la synchronisation *active* (courbes C0 P0), figures 2a, 3a et 4a, qui montrent des résultats catastrophiques quand le buffer circulaire tient dans le cache L1 ou L2. Au delà, la latence imposée par le cache L3 et la mémoire vive force la préemption des processus, ce qui libère du temps d'exécution pour les autres processus et viennent débloquer le processus en attente. À l'opposé, les meilleures performances sont

obtenues lorsque les processus sont assignés deux à deux sur les cœurs hyperthreadés, en privilégiant le regroupement en premier lieu d'un producteur avec le consommateur. La figure 2a présente ainsi une performance doublée pour la configuration *C0 P4* par rapport à l'assignation par défaut *C* P**. De façon générale, la stratégie *yield* présente un profil équilibré, peu importe les assignations, ce qui suggère une meilleure portabilité sur des systèmes variés. De plus, ces résultats sous-entendent une utilisation plus équitable des ressources de la machine que celle de la stratégie *active*, ce qui devrait s'avérer comme un atout sur une machine dont les cœurs sont fortement sollicités par d'autres tâches, comme la génération des données.
Pour conclure nos expériences, il est à noter que plus il y a de producteurs, plus les synchronisations *active* et *yield* offrent des performances similaires (hors assignation de tous les processus sur un même cœur).

4. État de l'art

La relation producteur - consommateur est explorée depuis de nombreuses années sous diverses formes de communication, de synchronisation et de gestion en mémoire des données échangées. Majoritairement, ces échanges de données entre processus utilisent la logique *FIFO* [7, 5] qui dans le cas de données ordonnées empêche l'écriture anticipée de données ultérieures et donc les bénéfices mentionnés en section 2.
Le stockage en mémoire des données échangées existe sous de nombreux termes qui désignent le même concept de *bufferisation* mais avec des avantages différents. Répondant à la notion de *queue* et de *liste chaînée* [7] pour les plus flexibles, et de *circular queue*, *circular buffer*, *ring buffer* [6] ou *double buffering* [10] pour l'utilisation d'une zone mémoire fixe pouvant être optimisée pour les interactions d'entrées/sorties grâce aux mécanismes de *mémoire mappée* ou de moteurs *DMA* [11]. Plus récemment, la stratégie *Bip Buffer* [2] vise à garantir l'accès en écriture et en lecture de zones mémoires contiguës par l'utilisation d'un pointeur d'écriture qui est replacé en début de buffer lorsque celui-ci n'a plus la place nécessaire pour stocker la prochaine donnée de façon contiguë. La stratégie présentée dans ce papier se rattache aux mécanismes basés sur une zone mémoire fixe.
Complémentaires au stockage des éléments en mémoire, les mécanismes de synchronisation des accès pour l'écriture et la lecture génèrent une attention particulière de par leur criticité dans les performances résultantes. Notre solution se base sur les principes de *lock free* [1, 3, 6] et *compare-and-swap* de données [8] qui permettent de réduire le coût de la synchronisation en évitant les coûteuses opérations de verrou proposées par les *mutex* et *futex* [9].
Parallèlement, de nombreux travaux s'intéressent à l'échange efficace de données dans des scénarios différents de celui traité dans ce papier : producteur et consommateur uniques [7, 5] ; producteurs et consommateurs multiples [4].

5. Conclusion

Dans ce papier, nous avons présenté une solution fonctionnelle et efficace de gestion des échanges non bloquants de données ordonnées de taille fixe entre un ou plusieurs producteurs et un unique consommateur. Notre solution, qui dispose de deux stratégies de synchronisation différentes, est jusqu'à 6,6 fois plus performante qu'une solution avec *mutex/cond* pthread et peut être améliorée davantage en assignant les processus sur les cœurs appropriés. Le design général de notre solution la rend applicable à de nombreux cas d'usage, qu'ils soient parallèles et/ou distribués. Nos prochains travaux visent à lever la contrainte sur la taille fixe des données pour permettre l'échange de données de taille variable, telles que des données compressées.

Bibliographie

1. Barnes (G.). – A method for implementing lock-free shared data structures. 1994.
2. Cooke (S.). – The bip buffer - the circular buffer with a twist, 2014.
3. Fraser (K.) et Harris (T.). – Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, vol. 25, n2, 2007, p. 5.
4. Gidenstam (A.), Sundell (H.) et Tsigas (P.). – Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. – In *International Conference On Principles Of Distributed Systems*, pp. 302–317. Springer, 2010.
5. Lê (N. M.), Guatto (A.), Cohen (A.) et Pop (A.). – Correct and efficient bounded fifo queues. – In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pp. 144–151. IEEE, 2013.
6. Lee (P. P.), Bu (T.) et Chandranmenon (G.). – A lock-free, cache-efficient shared ring buffer for multi-core architectures. – In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 78–79. ACM, 2009.
7. Torquati (M.). – Single-producer/single-consumer queues on shared cache multi-core systems. *arXiv preprint arXiv:1012.1824*, 2010.
8. Valois (J. D.). – Lock-free linked lists using compare-and-swap. – In *PODC*volume 95, pp. 214–222, 1995.
9. Wagner (T.) et Towsley (D.). – Getting started with posix threads. *University of Massachusetts at Amherst*, 1995.
10. Whitton (M. C.). – Memory design for raster graphics displays. *IEEE Computer Graphics and Applications*, vol. 4, n3, 1984, pp. 48–65.
11. Zinner (C.) et Kubinger (W.). – Ros-dma: a dma double buffering method for embedded image processing with resource optimized slicing. – In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pp. 361–372. IEEE, 2006.